Controlling and sequencing an heavily pipelined floating-point operator

André Seznec, Karl Courtel

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1554

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

CONTROLLING AND SEQUENCING AN HEAVILY PIPELINED FLOATING-POINT OPERATOR

André SEZNEC
Karl COURTEL

Novembre 1991

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Controlling and sequencing an heavily pipelined floating-point operator*

André Seznec, Karl Courtel†
IRISA, Campus de Beaulieu
35042 Rennes Cedex
FRANCE
e-mail : seznec@irisa.irisa.fr

October 7, 1991
Publication Interne n° 614 - Octobre 1991 - 28 pages - Programme I

CONTROLE ET SEQUENCEMENT D'UN OPERATEUR FLOTTANT FORTEMENT PIPELINE
Résumé

OPAC est un prototype d'accélérateur flottant développé fortement pipeliné à l'IRISA entre 1988 et 1991. OPAC a été con cu comme la cellule de base d'un coprocesseur dédié a' l'exe'cution des noyaux "compute-bound".

Notre objectif de performance était d'atteindre des performances proches d'une multiplication-accumulation flottante par cycle et par cellule sur des applications telles que résolution de système linéaire, FFTs, corrélations, ..

Les performances des opérateurs pipelines sur les applications scientifiques souffrent d'un temps de start-up élevé, particulièrement sur les boucles ayant un nombre d'itérations limité.

Dans cet article, nous présentons les mécanismes matériels originaux de controle du pipeline et de séquencement que nous avons mis en oeuvre sur le prototype d'OPAC. Ces mécanismes permettent d'obtenir des performances proches des performances asymptotiques sur de nombreuses applications et ceci, en particulier sur des boucles où le nombre d'itérations est inconnu à la compilation et peut être petit.

*This work was partially supported by the French ministry of defense under grant DRET-INRIA No 88.34.191.00.470.75.01 and the CNRS (PRC-ANM and GCIS)

†at present, BULL S.A., Les Clayes-sous-Bois

Abstract

OPAC is a prototype of a heavily pipelined floating-point operator which has been developed at IRISA during years 1988-1991. OPAC was designed as the basic of multi-cell floating-point coprocessor dedicated to the execution of "compute-bound" kernels.

The goal was to reach performance close to its potential floating-point multiply-add per cycle on applications such as solving linear systems, FFTs, correlations, ..

Generally, performance of heavily pipelined processors on scientific applications suffers from a very large start-up time, particularly when loops to be executed have a limited number of iterations.

In this paper, we present original hardware mechanisms for controlling and sequencing on the OPAC prototype. These hardware mechanisms allow to obtain near asymptotic performance on many applications and this even when the numbers of iterations in loops are unknown at compile time and may be small.

Keywords

pipelined operator, numerical application, hardware pipeline management

1 Introduction

OPAC is a prototype of pipelined floating-point operator which has been developed at IRISA during years 1988-1991. The architecture of the OPAC operator is justified elsewhere [SEZ91, COU91] (figure 1)). The OPAC operator has been designed in order to be the basic cell of a multi-cell floating-point coprocessor dedicated to the execution of compute-bound kernels (figure 2).

Our goal was to reach performance close to one floating point multiply-add per cycle and per cycle on effective applications such as solving dense linear systems or eigenvalue problems.

In application domains such as signal processing, image processing or numerical modelization, many algorithms may be written in such a way that the major part of the computations are encapsulated in calls for compute-bound kernels such as matrix updatings or primitives of BLAS level 3 library [DON88] as in LAPACK [AND90] : as most of the computers reach near optimal performance on such kernels, using optimized libraries becomes a de-facto standard.

But when implementing numerical libraries such as BLAS 3, parameters (i.e. size of the arrays) for the call are not known at compile time : achieving high performance for all the possible parameter values is a challenge for the architect of a pipeline processor.

In this paper, we present and justify the hardware mechanisms we have imagined and implemented for controlling the pipeline and for sequencing loops on the OPAC prototype.

In section 2, we give a glance at the OPAC genesis and we explain why performance on loops with a small number of iterations is a critical issue for global performance on OPAC.

In section 3, we recall some software problems associated with the use of horizontal microcode and classical sequencer. In section 4, the pipeline microcode used for controlling the computation block in OPAC is introduced; then section 5 presents the original mechanisms used in the OPAC sequencer to sequence short loop bodies.

We focus our attention on the hardware management of a heavily pipeline structure; our implementation allows to reach near nominal performance on most of the compute-bound kernels even when iteration numbers are small.

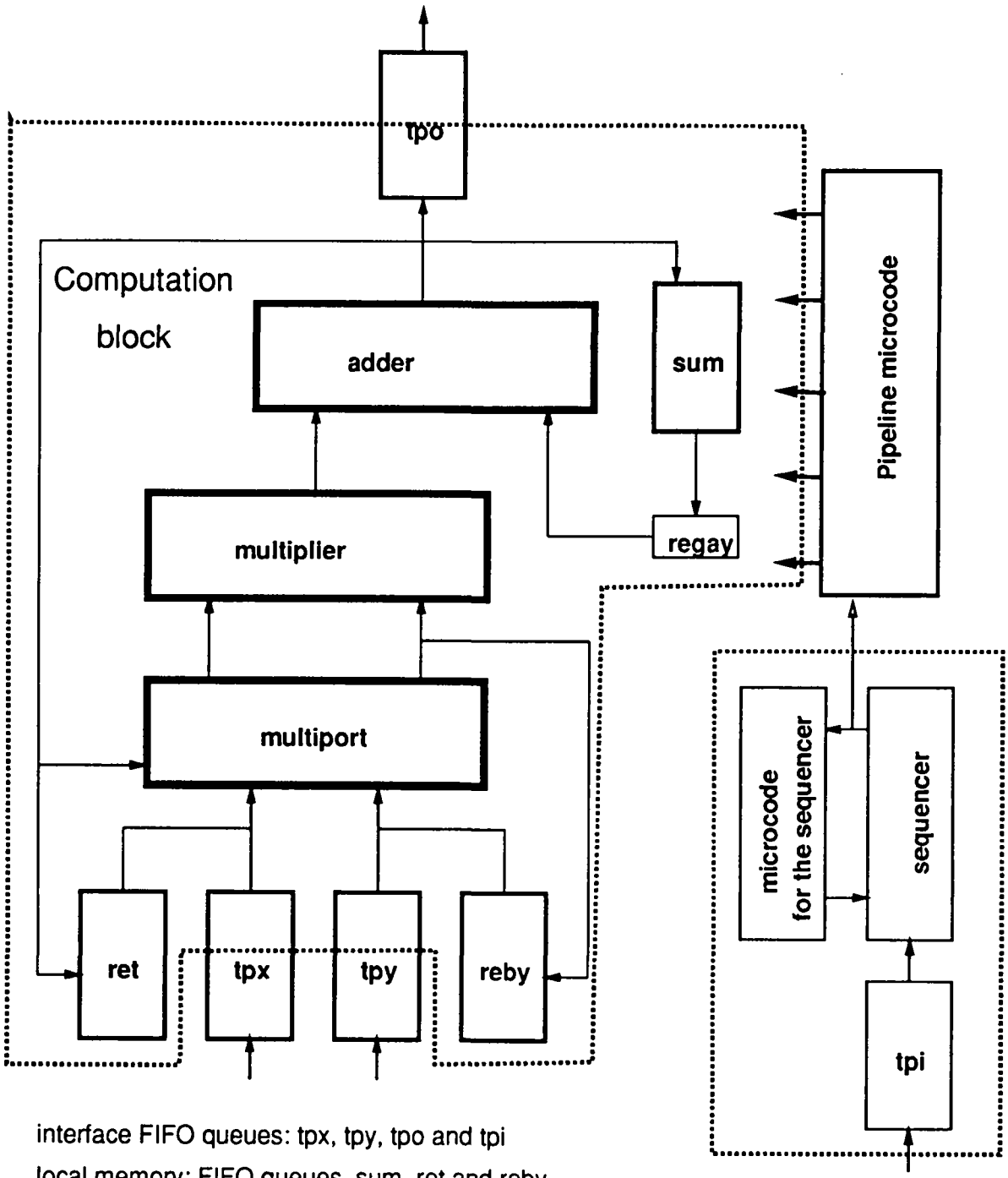
2 Presentation of OPAC

2.1 The OPAC genesis

When the architecture of the OPAC operator (figure 1) was specified, we tried to solve several problems :

1. We wanted *effective* performance close to one floating-point multiply-add per cycle on calls to the most frequently used compute-bound kernels (matrix updating and the whole BLAS 3 library, FFTs, correlations,..) and these kernels had to be library routines : the parameter values and particularly the size of the arrays are not known at compile time.

Figure 1: OPAC architecture



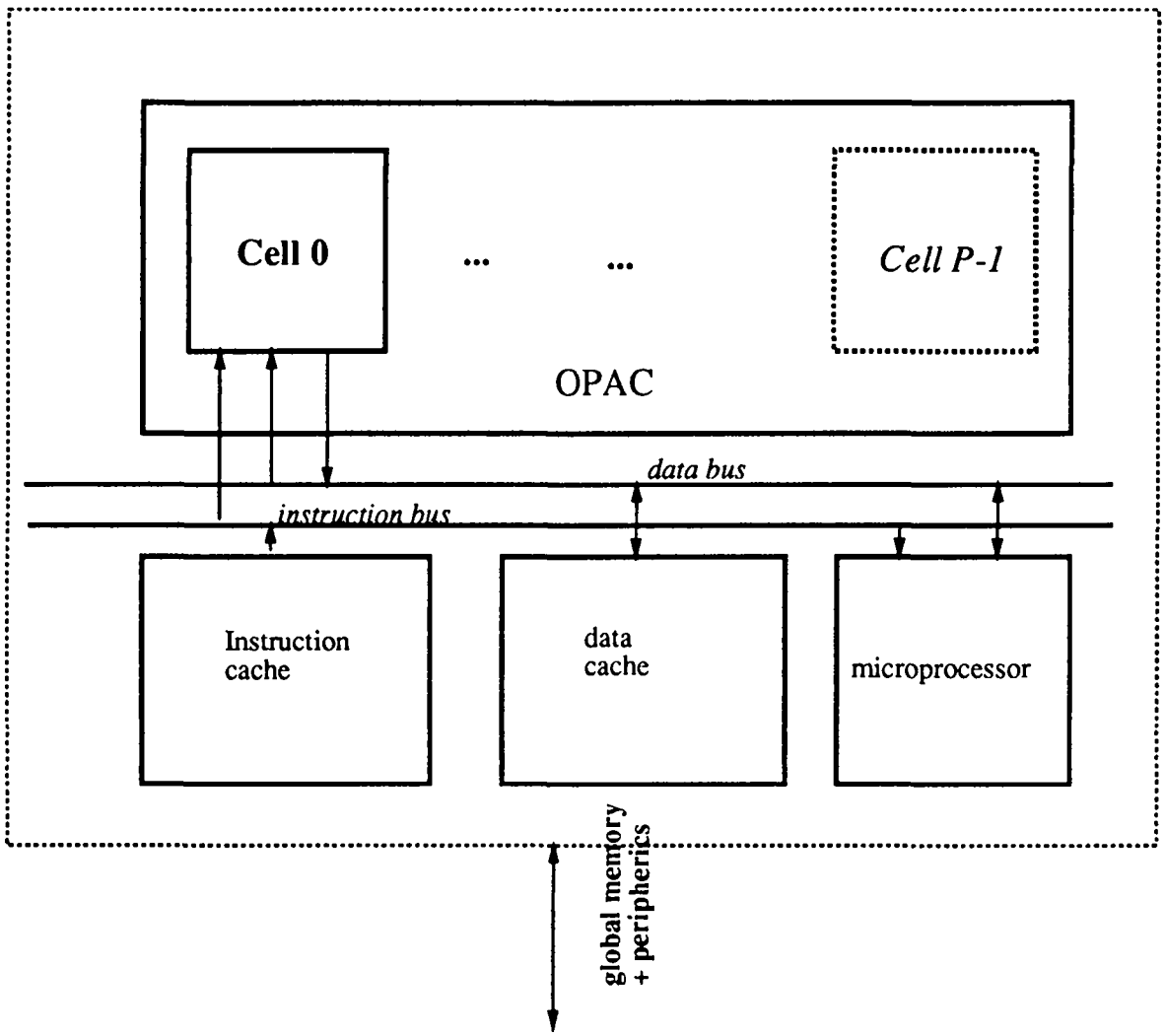
interface FIFO queues: tpx, tpy, tpo and tpi

local memory: FIFO queues sum, ret and reby

registers : multiport, regay.

operators: floating–point multiplier and adder

Figure 2: a multi-cell OPAC coprocessor in a microprocessor environment



2. We had to design a floating-point operator with a reasonably large local multiport memory with dynamic addressing [SEZ91] : the data bandwidth on the connection host with multi-cell coprocessor (figure 2) is quitre limiited.

3. We were architects (but not really software guys) but we had to generate efficient code.

OPAC architecture is illustrated figure 1. In the OPAC operator, FIFO queues are used as local memories : not just buffers, but effective dual-port memories with implicit addressing. Needs for a multiport local memory of several Kwords has been justified in [COU91, SEZ91].

Most of the compute-bound primitives have the following structure : nested loops which inner loop body consists in the evaluation of one (or a very few) arithmetical expression(s), often a single floating-point multiply-add. In order to limit the number of primitive calls on OPAC, the sequencer on OPAC is able to manage kernels with a relatively complex structure : three nested loops are needed to implement kernels such as matrix multiplies or FFTs.

Here we enumerate the steps of the genesis of the architecture represented figure 1.

1. The performance goal is unreachable without pipeline technic : OPAC is heavily pipelined.

2. Systematically chaining the floating-point multiplier and the floating-point adder does not induce significant loss on performance on the targeted compute-bound kernels.

3. We observe that all the targeted compute-bound kernels may be implemented using a local multiport memory where only one-strided address computation was needed : but at least three ports would be wished on this local memory.

4. Then we remark that this local memory may be replaced by hardware FIFO queues :

   - A FIFO queue may be used as an effective dual-port memory with dynamic but implicit addressing.
   - Address generation is simplified : no data path is needed to initialize the address generator.

   In some algorithms, three FIFO queues were needed (FFT and evaluation of a polynom on a vector).

5. Asynchronous sequencing between the multi-cell OPAC coprocessor and its host is used : when the sequencing on the host is stalled by a cache miss, a TLB exception or a page fault, the sequencing on the operator may go on until there are some data miss on the interface FIFO queues.

   Autonomous sequencing of each OPAC operator has been chosen : synchronous control of the cells would have lead to the development of two separate hardware entities : the global control unit and the basic cell.

6. We chose the execution of a compute-bound kernel as task granularity on the OPAC operator :

   the task granularity had to be relatively large in order to limit the throughput demanded on the connection OPAC-host (figure 2.

7. How to control and sequence on this operator and to obtain *effective* performance is the subject of this paper.

2.2 Pipeline management : a critical issue for performance on OPAC

Let us consider the matrix updating on OPAC. In order to limit the demand on exchange of data from and to the host, intermediate results and reusable are stored in the internal FIFO queues in OPAC as shown in the algorithm illustrated in figure 3. In this implementation, the whole updated matrix A M*N is stored in the FIFO queue sum in OPAC. If M*N exceeds the size of the FIFO queue sum then a block matrix updating algorithm is used, sizes of the blocks are chosen such that a block of A fit in the FIFO queue *sum*.

Analysis shows that using square blocks will limit data exchanges between the host and OPAC. On the current OPAC prototype, the size of the FIFO queues is 2048 words : square root = 45; in a VLSI implementation of OPAC, this size would be probably limited to 512 words (or less) : square root = 22.

For most of the compute-bound kernels we want to implement on OPAC, the same situation arises : due to the finite size of the FIFO queue, we need to implement block versions of the kernel.

But most of the computations lie in the inner most loops, and on OPAC the iteration numbers of these innermost loops will be quite limited (less than 50 for our prototype, in the order of 20 in a VLSI implementation) : in order to achieve effective performance close to one floating-point multiply-add per cycle on the OPAC operator, this order of performance has to be reached when sequencing loops with very limited iteration numbers.

In the next section, we show that this is not an easy challenge on pipeline processors : start-up delays may dramaticaly decrease performance.

3 Horizontal microcode pipeline processors

The OPAC prototype has been implemented with 1988 off-the-shelf components. In order to achieve high performance, all the functional units in OPAC must be activated on each cycle; then to control this functional units, it is necessary to apply them many bits of controls on each cycle. We chose to deliver these control vectors from a microcode memory.

Principles of the pipelined control of OPAC are derived from the control by horizontal microcode [RAU81, RAU82]. Nevertheless on OPAC, the implementation of control has been

Figure 3: Sequencing a matrix update on OPAC and its host

Sequencing on OPAC :

1) Initialization :

Load of A in the FIFO queue sum
from FIFO queue tpx

2) Computing :

```
For k=1 to K do
Load of vector B(. ,k) in  FIFO queue reby
                           from FIFO queue tpy

   For n=1 to N do
     For m=1 to M do
        A(m,n)= A(m,n)+ B(m,k)*C(k,n)
     Endfor
   Endfor
% A(m,n) is read and written on FIFO queue sum
% B(m,k) is read and written on FIFO queue reby
% Constant C(k,n) is read on FIFO queue tpx

  Reset of FIFO queue reby

Endfor
```

3) Sending results to the host

FIFO queue sum is emptied in FIFO queue out

Sequencing on the host:

0) sending the call for OPAC
and parameters M,N,K

1) Initialization:

Store matrix A in FIFO queue tpx

2) Computing:

For k=1 to K do

Store B(.,k) in FIFO queue tpy

Store C(k,.) in FIFO queue tpx

Endfor

3) Store the result matrix

slightly modified in order to solve some problems by hardware which are generally lead to software on classical horizontal microcode processors.

3.1 Horizontal microcode and code generation

In synchronous pipelined processors such as the FPS 164 from Floating-Point Systems [Ch81, TOU84] or the VLIW processors [COL88], an instruction is issued on every cycle. In this instruction, there is an instruction parcel for each functional unit in the processor; these instruction parcels are executed on the following cycle.

Performance that can be obtained on this family of processors depends on the quality of the executed microcode. Software technic for generating efficient microcode have been developed in the past decade [ELL85]. The most popular technic is the so-called software pipeline.

We describe here the particular steps for microcode optimization :

1. Microcode compaction :
   This step is characteristic to horizontal microcode generation. Several micro-instructions may be initiated at the same time on the different functional units. Compacting microcode consists in minimizing the number of instructions necessary to code a linear sequence. Constraints must be respected :

   (a) In order to respect the semantic of the program, a partial ordering of the micro-operations is defined by data dependencies; moreover delays imposed by the architecture between dependent micro-instructions must be respected.

   (b) Sharing the resources is a quite difficult problem :
   When several independent operations require the functional units, the code generator has to chose which of these operations has to be issued first.

   There may be conflicts on resources shared by several functional units (for example buses or registers).

2. Loop optimization [EIS88] :
   The target domains of most of the horizontal microcoded machines are the scientific applications and the signal and image processing.

   In these application domains, most of the computations lie in loops such as scalar dots or very short loop bodies. To reach correct performance when executing such loops on an heavily pipelined processor, several iterations must be executed concurrently.

   In order to achieve performance on horizontal microcoded processors, software techniques have been developed as e.g. the software pipeline [LAM87, RAU81] :

   One looks for a reservation table assoociated with a single iteration that allows a minimum delay L between the initiation of two consecutive iterations This reservation table leads to generate a "derived" loop body of L instructions where D/L consecutive iterations are in progress (D is the length of the reservation table).

A prelude must be added to enter the "derived" loop : the pipeline must be filled by the first iterations. And a postlude is added to empty the pipeline.

This method allows to overlap the execution of the iterations of the loop and generally to use the maximum parallelism available in the machine during the stable phase of the pipeline.

Nevertheless a minimum number of iterations is needed in the original loop in order to reach the first iteration of the "derived" loop body : particular cases must be managed by another way. Performance may then be poor when the iteration number is limited : time wasted during iteration tests, prelude and postlude does not depend on the iteration number.

3.2 Minimum loop body size

In numerical applications, most of the computations lie in loops. In most cases, the loop bodies consist in evaluating a very few arithmetic expressions : only one floating-point multiply-add in dot products or internal loops in matrix multiplies.

Many horizontal microcoded machines are able to initiate one floating-point multiply and one floating-point add per cycle. On many machines, software pipeline technic would allow to obtain a loop body consisting in a single instruction word. Unfortunately generally classical sequencers cannot manage single instruction loops but this cannot be used if the sequencer is able to sequence loop bodies consisting in only one instruction.

Off-the-shelf sequencers (e.g. AMD-2910, ADSP-1401, CY7C910) are designed around the same type of architecture. They consist in a "program-counter" which may be updated with different values flowing out from different sources : a stack for routine management, a counter for loop management or microcode memory. The instruction flows out from a microcode memory.

Then managing the sequencing of a loop consists in several consecutive steps :

- Read of the loop management instruction in the microcode memory
- Decrementation and test on the counter associated with the loop
- Branch on the "program-counter"

If all these steps can be executed in a single cycle then loops with bodies consisting in a single instruction may be sequenced. But in this case, the latency of the pipeline described here may be the critical path for determining the machine cycle.

In most current implementations of horizontal microcoded machines, this latency is equal to 3 or 4 cycles.

A software solution consists in unrolling loop bodies in order to obtain a new loop body longer than the previous latency.

3.3 Limits of the software pipeline management

In this section, we show the limits in terms of performance of the software pipeline management when the iteration number is limited. We illustrate this on a hypothetic operator OPAC controlled by horizontal microcode and for which the minimum size of a loop body is 4 instructions.

Let us consider now the two-fold nested loop that follows (it is the computation of the solution of a triangular system) [1]:

```
for i = 1 to I
      for j= i to I
          A[j]= A[j]- B[i,j]*C[i]
      endfor
endfor
```

Let M= I-i+1.

The delay for crossing the whole pipeline in OPAC is 12 cycles.

Let us consider software pipeline generated by this loop :

- 15 iterations at minimum are needed to reach the first iteration of the "derived" loop : 12 iterations are needed to fill the pipeline and (4-1) are needed to the first iteration of the "derived" loop.
- if $M \geq 15$ then the remaining (M -15) mod 4 iterations cannot be sequence in the "derived" loop.
- The particular cases where $M < 15$ must be treated.

In order to maintain the code volume in acceptable limits and to keep a reasonable software complexity for the code generator, horizontal microcode generators do not generally optimize the particular cases. In our example, the number of instructions necessary for coding the fourteen particular cases would be $\Sigma_{i=1}^{15}(11 + i) = 259$ instructions plus the instructions needed for the branching sequence.

One generally prefers to use completely sequential execution for particular cases.

Then the minimum sequencing time of iteration i of the outer most loop in the example is given by formula (1) :

$T_{test} + (M < 15) * M * 12 + (M \geq 15) * (((M - 11)/4) * 4 + T_{prel} + T_{post} + T_{test} + (12 * (M - 11\ mod\ 4))) + T_{loop}$ cycles

where $T_{test} = 4$ is the delay needed for a test (here $(M < 15)$ and ((M-11) mod 4 = 0)),
T_{prel}=11 is the execution time for the prelude,
T_{post}=11 is the execution time for the postlude
T_{loop}=4 is the delay for managing the outer most loop.

[1] To prevent easy critics on our example choice, we have chosen to present results on the solving a triangular system because the number of iterations vary in the inner most loop

Formula (1) allows to compute the execution times for this loop for different values of I (see table 1). Only sequencing constraints are considered here; other constraints may increase the execution time, but in this paper, our aim is to analyze only the control and sequencing mechanisms.

We also give the efficiency of the computation (i.e the ratio of the number of floating-point multiply-adds on the number of needed cycles). This table clearly shows that asymptotic efficiency is very close to one, but that effective efficiency parameters is smaller : for I=300, the efficiency is only 0.8. Moreover as we pointed out in section 2.2, for applications on OPAC, block versions of the algorithms will be implemented and iteration numbers will be in the range of 20 to 50.

I	20	50	100	300	1000	2000
Nb cycles	1612	3764	9292	56492	536692	2072692
Efficiency	0.065	0.338	0.543	0.799	0.932	0.965

Table 1

In order to be able to achieve correct performance, we have imagined the hardware mechanisms for controlling and sequencing on the OPAC operator which are presented in the next sections.

4 Pipelined microcode

In the section 2, we have pointed out that three internal FIFO queues were needed in the OPAC architecture in order to be able to implement the most useful compute-bound kernels (e.g. FFTs).

When sudying the compute-bound primitives, we also noticed that these three FIFO queues may be specialized and have only one input path and one output path; this is also true for the interface FIFO queues (see figure 1).

Let us consider the set S of subfunctional units in the computation block composed by :

- Exits of the FIFO queues tpx, tpy, reby, ret, sum
- Entries of the FIFO queues ret, sum, reby, out
- Floating point ALU and floating-point multiplier
- Register file

The architecture of the computation block in OPAC has the following properties :

(a) the set S is partially ordered by the relation noted $<<$ generated by the precedence relation:
X precedes Y iff Y works on data flowing out from X

(b) There is at most one path from X to Y i.e. :
if $(X << X1 << Y)$ and $(X << X2 << Y)$ then $((X1 = X2)$ or $(X1 << X2)$ or $(X2 << X1))$

(c) for every component $s \in S$, its latency is a constant $Lat(s)$

(d) there is a path from every component S to the *final* FIFO queues ret, sum and out.

Then if a result has to be written on one the *final* FIFO queues ret, sum or out at cycle t, we can date the contribution of every component to this result :

the contribution of X was initiated at cycle $t - \Sigma_{Y \in PATH} Lat(Y)$ where PATH is the unique path from X to the final FIFO queues.

It is then natural to delay the different instruction parcels in order to store in the same instruction word, the instruction parcels working on the same data flow. This is illustrated in figure 4.

Then, all the actions in the computation block concerned with a single floating-point operation are coded in a single instruction word :

- Reads of FIFO queues
- Reads and writes of the register file
- The floating-point operation (it may be a multiply-add)
- Write(s) of the result in the "final" FIFO queue(s)

For code generation, the computation block may be considered as a monolithic combinatorial circuit which would be crossed in a single cycle : the pipeline latency is completely hidden and there is no need for using software pipeline technique in order to reach asymptotic performance on OPAC.

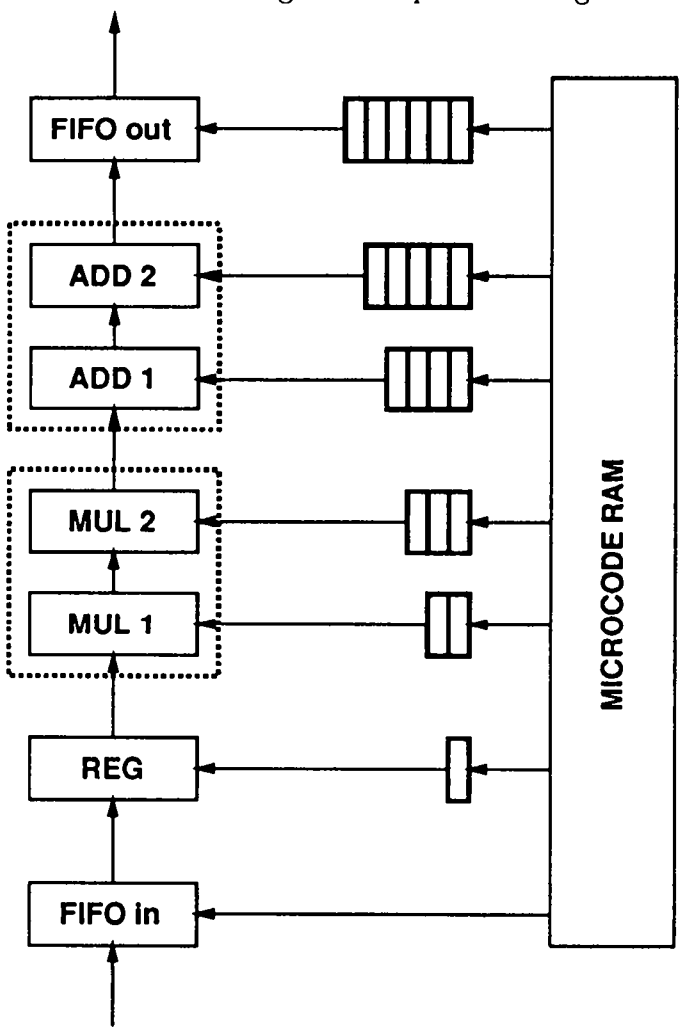
As no particular case has to be studied and no prelude and postlude has to be generated, microcode volume is quite limited.

Moreover, this technic allows to overlap at execution time, the execution of two successive loops (e.g.two iterations of an outer most loop) without wasting any time for executing prelude and postlude of the loops.

We analyze the benefit of the pipelined microcode on the example considered in section 3.3.

As in the previous section, we consider that the minimum length of a body is 4 cycles. To achieve asymptotic performance of one floating-point multiply-add per cycle, we must unroll the internal loop in order to obtain an internal body loop with 4 floating-point multiply-add.

Figure 4: Pipeline management in the computation block



```
for i =1 a I

    for j1= 0 to ((I-i)/4)-1
        A[4*j1+1+i]= A[4*j1+1+i]-B[i,4*j1+1+i]*C[i]
        A[4*j1+2+i]= A[4*j1+2+i]-B[i,4*j1+2+i]*C[i]
        A[4*j1+3+i]= A[4*j1+3+i]-B[i,4*j1+3+i]*C[i]
        A[4*j1+4+i]= A[4*j1+4+i]-B[i,4*j1+4+i]*C[i]
    endfor
    for j2= 1 to (I-i) mod 4
        A[i-j2+1]=A[i-j2+1]-B[i,i-j2+1]*C[i]
    endfor
endfor
```

The internal loop body may be coded on 4 instructions.

The execution time of one external loop iteration is decomposed in :

1. (I-i)/4 = 0 ? : 4 cycles
2. loop on j1 : 4* ((I-i)/4) cycles
3. j2 = 0 ? : 4 cycles
4. loop on j2 : ((I-i) mod 4)* 4 cycles
5. loop management on i : 4 cycles

Performance induced by the sequencing constraints is represented in table 2. The efficiency obtained when using horizontal microcode and software pipeline is recalled in row SP.

I	20	50	100	300	1000	2000
Nb cycles	540	2100	6700	50100	517000	2034000
Efficiency	0.194	0.607	0.753	0.901	0.968	0.984
SP	0.065	0.338	0.543	0.799	0.932	0.965

Table 2

In this table, we clearly observed the improvement in performance for small parameter values obtained by using the pipelined microcode versus the classical horizontal microcode.

In terms of hardware, the cost of the pipelined microcode consist in only registers to delay the microcode. In the case of a VLSI implementation, this cost would be quite insignificant. Moreover, the microcode volume will dramatically decrease.

5 Specific functionalities in the OPAC sequencer

We have chosen to implement an ASIC sequencer for OPAC. In this section, we describe the special features of this sequencer :

- Hardware management of interruptions
- Optimal loop management

5.1 Interruptions on OPAC

The execution of a floating-point operation on OPAC may be conditioned to the presence of the operand data in FIFO queues. These FIFO queues may be internal FIFO queues (reby, sum and ret) or interface FIFO queues (tpx and tpy). When some data operand lacks on a FIFO queue, the sequencing has to be interrupted.

Interface FIFO queues

Let us remark that there is a specific difficulty when using off-the-shelf FIFO queue components : these FIFO queue components have a "Empty flag", but the behavior of this flag does not allow to use it to obtain a sustained throughput of one word per FIFO queue cycle.

> When a read is done at cycle T, the new "Empty Flag" induced by this read is available only at the end of the cycle. But the decision to read or not the FIFO queue at cycle T+1 has to be taken at last during cycle T.

There are approximately three solutions to be able to achieve a potential throughput of one word of data per cycle on a FIFO :

1. Use of a longer clock : the period must be include the FIFO queue cycle plus the delay to take the decision to sequence or not the instruction.
2. The use of software technic to ensure the data presence.
3. The information "Empty Queue" is managed in the sequencer (e.g. by associating a counter/decounter with each of the FIFO queues). Let us remark that the effective interesting information is "WE ARE SURE THAT THE FIFO QUEUE WILL NOT BE EMPTY".

Guaranteeing by software the presence of data in one of the interface FIFO queue would have been difficult, so we have chosen to implement the management of the presence of the data in the interface FIFO queues at the sequencer level.

Internal FIFO queues

The presence of a data in the internal FIFO queues may be guaranteed by software : length of the pipelines are known. This will lead to software management of particular cases as presented below :

Let us consider the same example as in the previous sections :

```
for i = 1 to I
    for j= i to I
        A[j]= A[j]- B[i,j]*C[i]
    endfor
endfor
```

The natural way to limit data exchanges with the host in this example is to use the FIFO queue sum as intermediate storage for the vector A.

In the OPAC design, there must be a minimum 6 cycles delay between two successive updates of the *same* data if it has to be read on FIFO queue sum, updated and back stored in FIFO queue sum. When the number of elements in the vector to be updated A is inferior to 6, the new update of a data must be delayed until the data becomes accessible in the FIFO queue.

Managing this by software is possible, but would increase the microcode size and decrease the performance : test of iteration numbers, particular case treatment ,..

In order to avoid these difficulties, we have chosen to manage the presence of data in the internal FIFO queues by hardware in the sequencer.

Hardware interruption management in OPAC

On many hardware implementations where pipeline processors are interfaced through FIFO queues, the solution consists in stalling the whole pipeline when a data lacks on a FIFO queue.

Such a solution would not have permitted to manage by hardware the miss on an internal FIFO queue; if the whole pipeline is stalled then no data can be written on the faulting FIFO queue and the miss will never disappear : deadlock.

The principle of interruption management in OPAC is illustrated in figure 5.

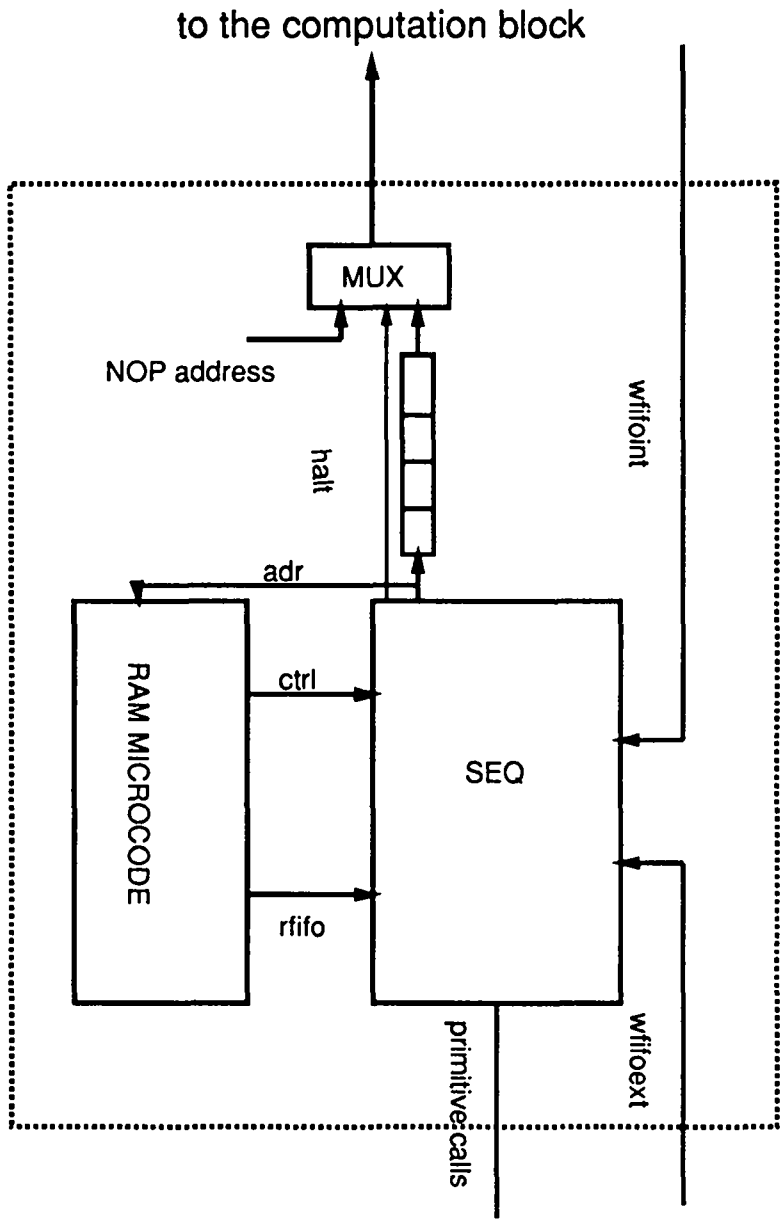
In the sequencer, a counter is associated with each of the FIFO queue. This counter uses READ information (a bit in rfifo) and WRITE information (one bit in wfifoext for tpx and tpy, one bit in wfifoint for sum, reby and ret).

The successive actions to sequence an instruction at address Ad are :

1. Ad flows out of the sequencer, but does not flow directly to microcode RAM controlling the computation block.

2. The word at address Ad in the RAM microcode associated with the sequencer

Figure 5: Principles of the interruption management in OPAC



3. The instruction parcel rfifo is use to determine whether the instruction can be issued in the computation block or not : the parcel rfifo contains one bit for each of the FIFO queue indicating whether the FIFO queue will be read or not during the instruction.
   When the instruction will possibly cause a miss, the sequencer issues a HALT flag which freezes the pipeline in the sequencing part of the operator (i.e the part illustrated in figure 5).
   But the pipeline in the computation block (see figure 1) is not frozen : address of a NOP (no operation) is sent, and instructions which have been already been initiated in the computation block go on in the pipeline. Then, if the interruption is due to a miss on an internal FIFO queue, unless program error the data missing would be written by an instruction already in the pipeline.

This hardware management of the misses in internal FIFO queues has been feasible because of the structure of the pipeline in the computation block and its pipelined control.

5.2 Optimal loop management

As pointed out in the previous sections, the use of classical sequencers leads to a minimum number of instructions in loop body of the order of 3 or 4. On the other hand, in a lot of numerical applications (e.g dot product), the source code of the inner loop body consists in a floating-point multiply-add and would naturally lead to a single instruction body loop microcode.

We present here the principles of an hardware mechanism for managing loops with very short loop body -number of instructions must be known at compile time.

We have called this mechanism "by-clock decrementation".

Principle :

We use :

- Two decounters C_{inst} and C_{it}
- One register R_{inst} associated with C_{inst}
- One address register A_{it} associated with C_{it}

Let us consider that :
-C_{inst} and R_{inst} are initiated with the number of instructions in the loop body.
-C_{it} is initiated with the number of iterations of the loop.

Then the whole loop is managed by initiating this unique instruction :

while ($C_{it} \neq 0$) On clock {
    C_{inst}:=C_{inst}-1;
    if $C_{inst} \neq 0$ then
        PC:=PC+1
    else {
        C_{it}:= C_{it}-1; C_{inst}:= R_{inst};

```
            if C_it =0 then
                        PC :=PC+1;
            else
                        PC := A_it;
        }
}
```

This instruction may seem complex, but it may be implemented very simply in hardware and it even may be pipelined (figure 6) :

-the instruction is initiated by a stimuli STI coming back from the microcode, it is executed until $C_{it} = 0$: this is managed by the decrementation manager DEC.
-the different steps of the instructions may be pipelined :

- the action on the program counter for example may depend on what has happened on the previous cycle.
- Value 1 of decounter C_{inst} may be detected and then the transition of C_{inst} to 0 may be anticipated in order to decount on C_{it} and to reinitialize C_{inst} with R_{inst}.

The critical electrical path in this mechanism is a decounter decrementation.
This hardware mechanism allows to sequence a loop with a body consisting in $Inst$ instructions in $Inst * I + lat$ cycles where lat is the latency of the pipeline of the sequencer (Program Counter, Read of instruction, Decode, Initiating the stimuli) and I the number of iterations.

Managing nested loops

Let us consider of the two inner-most loop in a matrix update:

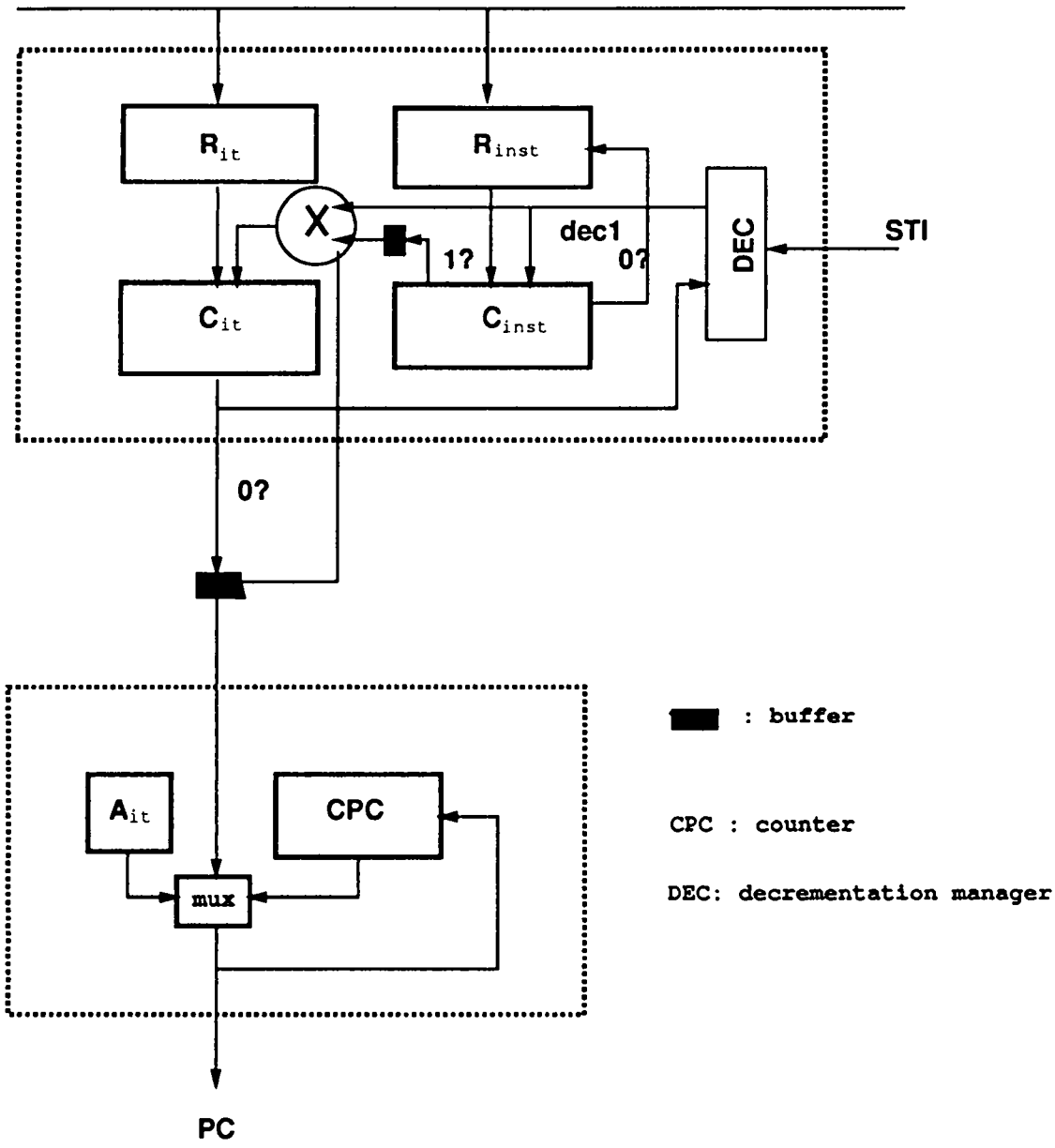
```
 for n = 1 to N


      for m= 1 to M
          A[n,m]= A[n,m]+ B[n,k]*C[k,m]
      endfor
endfor
```

If we use the previous mechanism to sequence the inner loop and a classical mechanism to sequence the external loop then the sequencing of one iteration of the outer loop will cost $M + 2 * lat$ cycles :

1. lat cycles for initiating the inner loop

Figure 6: Principles of the "by-clock decrementation"



2. M cycles of computation

3. *lat* cycles for managing the outer loop

As pointed out in section 1, the implementation of the matrix updating on OPAC will lead to a block algorithm where M will be of the order of the square root of the FIFO queue size i.e about 45 for the prototype :
if *lat* = 4 then the performance may be limited to about 45/53= 85 % of the theoretical floating-point multiply-add per cycle simply due to time wasted for initiating the inner loop and for managing the outer loop.

To avoid this loss in performance, we have extended the hardware mechanism presented above in order to manage nested loops, the inner loop may be managed by "by clock decrementation": a set of instructions derived from the instruction described above are implemented.

We use :

- a set of decounters C_i
- a set of registers R_i respectively associated with C_i
- a set of registers A_i respectively associated with C_i

An example of loop management instruction is given here for a two-nested loop where the inner most loop body size exceeds *lat* cycles.
-C_1 and R_1 are initiated with the iteration number of the inner most loop.
-C_2 is initiated with the iteration number of the second loop.

The two-nested loop is managed by issuing this instruction on each iteration of the inner most loop: we call this instruction "cascaded decrementation"

C_1:=C_1-1;
if $C_1 \neq 0$ then
    PC:= A_1
else {
    C_2:= C_2-1; C_1:= R_1;
      if $C_2 \neq 0$ then
          PC := A_2
    else
          PC:=PC+1
    }
}

If the size of the inner most loop body is smaller than *lat* cycles, then we can use the same instruction encapsulated in a "by-clock decrementation".

We suppose here that C_0 and R_0 are initiated with the number of instructions of the inner most loop body, then the two-nested loop presented at the beginning of the section may be managed this unique "by-clock cascaded decrementation" :

```
while (C2 ≠ 0) On clock{
    C0:= C0 − 1;
        if (C0 ≠ 0) then PC:=PC+1
        else {
            C1:=C1-1;
            if C1≠ 0 then
                PC:= A1
            else {
                C2:= C2-1; C1:= R1;
                  if C2 ≠ 0 then
                      PC := A2
                else
                       PC:=PC+1
                }
            }}
```

Hardware mechanisms to execute this family of instructions have been derived from the mechanism presented in figure 6.

For simplicity of implementation, we have chosen to decompose the loop management instructions in two separate instructions :

1. A configuration instruction is used for the set of decounters C_i and their associated branch registers A_i.

   For a given loop, if C_i is involved in the management of the loop, the stimuli for decounting on C_i may be the clock or the passage to 0 of another C_j.

   The same information may be used in order to configure the branch registers; for each counter C_i, this configuration is :

   -On the decrementation of C_i, if $(C_i \neq 0)$ then $PC = A_i$

   or

   -On the decrementation of C_i, nothing with A_i (when C_i is used for "by clock decrementation" of the inner loop).

   When defining OPAC, we remarked that managing three loop levels is sufficient to be able to execute most of the useful compute-bound kernels. Then we implement 4 counters C_i; the information for configuring one counter and its associated branch register is coded on 4 bits, then the whole set of C_i may be configured by a single instruction.

   This instruction is used only one times to prepare the loop.

2. "decrementation initiation" instructions :

- if the size of the inner most loop is larger than *lat* instructions a "cascaded decrementation" is issued during execution of each inner loop iteration
- otherwise a single "by-clock cascaded decrementation" is issued before entering the loop : iteration of the inner most loop otherwise. In this instruction, the condition for stopping the "decrementation" is coded : passage to 0 of one of the decounters.

This hardware mechanism has been implemented in the ASIC sequencer of OPAC. As the mechanism presented in the section 5.2, it may be pipelined and the critical electrical path is a decounter.

With this mechanism the time needed for sequencing the two-nested loop previously considered is I*J+*lat* cycles.

6 Impacts of the controlling and sequencing mechanisms

6.1 Impacts on performance

On the OPAC prototype, the latency *lat* of the sequencer pipeline (Read of the microcode, decode, decrementation, program counter) is 7 cycles.

In table 3, we give the performance that could be obtained on the example considered in 3.3 when there are only constraints due to sequencing and controlling . The cost for sequencing the iteration i of the outer most loop is :

- 1 cycle for decrementing the number of iterations of the inner most loop
- 7 cycles to initiate the "by clock decrementation".
- (I-i+1) cycles

Rows SP and PM recall respectively the limits when using respectively only software pipeline and pipeline microcode without specific sequencer.

I	20	50	100	300	1000	2000
Nb cycles	265	1675	5850	47450	508500	2017000
Efficiency	0.396	0.761	0.863	0.951	0.984	0.992
PM	0.194	0.607	0.753	0.901	0.968	0.984
SP	0.065	0.338	0.543	0.799	0.932	0.965

Table 3

As for the particular case of OPAC, the inner most loop iteration number would be generally in the range of 20 to 50, the benefits of the new mechanisms we have introduced in the sequencer for managing loops clearly appear.

6.2 Impacts on code generation complexity

Let us consider here the example of the two-inner most loops of the matrix updating:

```
for n = 1 to N
      for m= 1 to M
           A[n,m]= A[n,m]+ B[n,k]*C[k,m]
      endfor
endfor
```

We suppose M and N in the range of 40 to 50 (but not known at compile time).
Let us look at the software complexity needed in order to achieve reasonable efficiency.

Horizontal microcode and software pipeline

If we only consider the software pipeline, as exposed in section 3 then the formula (1) in section 3.3 gives the number of cycles needed for sequencing the inner most loop.

Table 4 gives the efficiency of this technic for N=M= 40,..,43 on the two-nested loop.

N=M	40	41	42	43
Nb cycles	2960	3524	4116	2838
Efficiency	0.540	0.477	0.428	0.651

Table 4

This leads us clearly to unacceptable poor performance. Software technique may be used in order to improve this performance on very specific compute-bound kernel :

as the number of iterations M is constant, tests may be brought outside the loop and residual number of iterations (modulo 4) may be unrolled and compacted with the postlude. This lead to a sequencing time of $M + T_{post}$ cycles for an iteration of the outer most loop.

Then the efficiency reaches about 78 %.

Pipelined microcode only

The same optimizations as for the software pipeline will lead to a sequencing time of 4*((M/4)+1) cycles for an iteration of the outer most loop. This can be considered as near optimum, but at a large code expansion cost.

OPAC implementation

As mentioned earlier, the number of cycles needed to sequence in OPAC the two-nested loop will be N*M+7 cycles without any software optimization.

6.3 Impacts on hardware complexity

Paradoxicaly, the introduction of our pipeline management and control mechanisms has simplified the functionalities needed on the sequencer of OPAC.

If classical horizontal microcode control had been used, then instructions would have been needed in order to test the iteration numbers, to calculate modulos etc.

Moreover, our mechanisms allow to obtain performance with a very compact microcode; only 11 instructions for a completely optimized matrix multiply working for all parameter values of the previous two-nested loop :

- Initialization of address registers (2 instructions)
- Configuration of the set of decounters (1 instruction)
- Initiation of the "by-clock decrementation" (1 instruction)
- 6 no operation instructions (the pipeline of the sequencer).
- the loop body (1 instruction).

This might be very important for a VLSI implementation of an OPAC-like operator.

6.4 "by-clock decrementation" in an other context

The hardware mechanism we have presented for managing loops may be of practical use in a microprocessor design.

When the number of instructions issued in each iteration of a loop is a fixed number known at compile time, a mechanism derived from the "by-clock decrementation" mechanism may be used :

- Generally loop management takes two instructions on current microprocessors : decrementation on a counter, (test and branch). Several instructions are issued on the same cycle on superscalar microprocessors in order to improve performance (e.g. up to four on the IBM RS6000). Using "by-clock decrementation" would allow to achieve the same level of performance on loops while potentially issuing one instruction less in parallel.
- On current microprocessor implementations, the minimum number of cycles per loop body depends on a similar pipeline as described in section 3.2 : considerable effort has been put to limit this latency by hardware (e.g. two cycles on the IBM RS6000). Using "by-clock decrementation" would partially remove the stress on this latency.

7 Conclusion

In many application domains, programmers are not specialists of parallel processing, but ask for performance. In signal processing, image processing or numerical modelization, many algorithms may be written in such a way that the major part of the computations are encapsulated in calls for compute-bound kernels. As most of the computers reach near optimal performance on compute-bound kernels, using optimized libraries becomes a de-facto standard.

Achieving high performance at a low hardware cost on the whole family of compute-bound primitives has became an important challenge; associating a specialized multi-cell floating-point coprocessor with a monoprocessor host may be cost-effective. On such a specialized floating-point coprocessor, the workload has to be distributed among the different cells, then achieving high performance on a compute-bound kernel for all parameter values depend on the ability to achieve near optimal performance on each cell on relatively limited sets of data.

OPAC has been designed to be the cell of such a specialized coprocessor. In this paper we have presented hardware mechanisms for controlling the pipeline and for optimally sequencing loops on the OPAC floating-point operator.

In section 2.2, we have explained why the reaching of near asymptotic performance on loops with a small number of iterations is a critical issue for reaching near optimal performance on the OPAC operator.

Classical structures of pipelines where functional units are not statically chained must be managed by software pipelines; but this technic does not allow to reach near nominal performance for short loops.

Although the OPAC operator is able to execute most of the useful compute-bound kernels, the functional units in OPAC are statically chained. The OPAC operator has been specifically designed in order to be controlled by a pipelined microcode which flows to the different functional units at the same speed that the data reaches them. Then for code generation, the whole computation block in OPAC may be seen as a combinatorial circuit responding in one cycle period.

In the sequencer, we have addressed the important problem of the loops with very short body (e.g. a single instruction). We have presented an original hardware mechanism which allows to optimally sequence nested loops and loops with short bodies (i.e. without losing cycles for loop management).

Our controlling and sequencing mechanisms allow to reach near nominal performance on the OPAC operator for small iteration numbers and without using complex software techniques for pipeline management. Moreover the microcode volume is very limited.

These hardware mechanisms have been implemented in a prototype of the OPAC operator. It has been realized with components which were off-the-shelf in 1988 and a customized ASIC sequencer (realized in Cmos 2μ). This prototype allows to validate the concepts presented in this paper.

The performance results allowed by our mechanisms may justify a VLSI implementation of an OPAC-like operator. The complexity of a VLSI implementation of an OPAC-like operator

has been evaluated to a million of transistors (512 32-bit words per FIFO queue).

References

[AND90] E.Anderson & al "LAPACK : A portable linear algebra library for high-performance computer" Proceedings of Supercomputing '90, New-York, Nov. 1990

[BOD89] F. Bodin "Optimisation de microcode pour une architecture horizontale et synchrone: Etude et mise en oeuvre d'un compilateur" Thèse de L'Université de Rennes I, Juin 1989

[COL88] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, P. Rodman, "A VLIW Architecture for Trace Scheduling Compiler", IEEE Transactions on Computers, Sept.1988

[COU91] "Etude et développement d'un coprocesseur de calcul matriciel: OPAC" K.Courtel, Thèse de l'université de Rennes I, Janv.91

[Ch81] A.E.Charlesworth, "An approach to scientific array processing : the architecture of the AP120B/FPS 164 Family" Computer, septembre1981

[DON88] J.Dongara, J.DuCroz, I.Duff, S.Hammarling "A set of level 3 Basic Linear Algebra Subprograms" Argonne Technical Report May 1988.

[EIS88] C.Eisenbeis, "Optimization of horizontal microcode generation for loop structures" 2^{th} International Conference on Supercomputing, Juillet 1988, St-Malo

[ELL85] , J.R. Ellis ,"Bulldog: A Compiler for VLIW Architectures", MIT Press, 1985

[JEG86] Y. Jégou, A. Seznec "Data Synchronized Pipeline Architecture : Pipelining in Multiprocessor Environment" Journal of Parallel and Distributed Computing Dec.1986

[LAM87] M. Lam,"A Systolic Array Optimizing Compiler", Ph.D. Thesis Carnegie Mellon University, May 1987

[RAU81] B.R. Rau, C.D. Glaeser, "Some scheduling techniques and easily schedulable horizontal architecture for high performance scientific programming" IEEE/ACM 14^{th} Annual Microprogramming Workshop Oct. 1981

[RAU82] B.R. Rau, C.D. Glaeser, R.L. Picard, "Efficient code generation for horizontal architectures; compiler techniques and architectural support" Computer Architecture News, Vol.10, Avril 1982.

[SEZ91] A.Seznec, K.Courtel "OPAC: a cost-effective floating-point coprocessor", IRISA report 586, May 1991

[TOU84] R.F. Touzeau, "A Fortran Compiler for the FPS-164 Scientific Computer", Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction, 1984

PI 604 A GENERAL METHOD TO DEFINE QUORUMS
Mitchell L. NEILSEN, Masaaki MIZUNO, Michel RAYNAL
Septembre 1991, 20 pages.

PI 605 OBTENTION DES EQUATIONS DYNAMIQUES D'UN SYSTEME PHYSIQUE A PARTIR DE SON MODELE BOND GRAPH
Bénédicte EDIBE
Septembre 1991, 26 pages.

PI 606 CONSTRUCTIVE PROBABILITY AND THE SIGNalea LANGUAGE : BUILDING AND HANDLING RANDOM PROCESSES VIA PROGRAMMING
Albert BENVENISTE
Septembre 1991, 60 pages.

PI 607 ABOUT LOGICAL CLOCKS FOR DISTRIBUTED SYSTEMS
Michel RAYNAL
Octobre 1991, 16 pages.

PI 608 UNE NOUVELLE APPROCHE REALISTE DE SIMULATION D'ECLAIRAGE DANS UN ENVIRONNEMENT DIFFUS
Eric LANGUENOU, Kadi BOUATOUCH, Pierre TELLIER
Octobre 1991, 70 pages.

PI 609 INTEGRATION D'UN CORRECTEUR ORTHOGRAPHIQUE DANS L'EDITEUR STRUCTURE GRIF
Patrice FRISON, Eric PICHERAL, Hélène RICHY
Octobre 1991, 22 pages.

PI 610 SYNCHRONIZATION AND CONCURRENCY MEASURES FOR DISTRIBUTED COMPUTATIONS
Michel RAYNAL
Octobre 1991, 20 pages.

PI 611 MALI v06 - TUTORIAL AND REFERENCE MANUAL
Olivier RIDOUX
Octobre 1991, 86 pages.

PI 612 SENSITIVITY COMPUTATION IN NETWORK RELIABILITY ANALYSIS
Gerardo RUBINO
Octobre 1991, 38 pages.

PI 613 OPAC : A FLOATING-POINT COPROCESSOR DEDICATED TO COMPUTE-BOUND KERNELS
André SEZNEC
Karl COURTEL
Octobre 1991, 28 pages.

PI 614 CONTROLLING AND SEQUENCING AN HEAVILY PIPELINED FLOATING-POINT OPERATOR
André SEZNEC
Karl COURTEL
Octobre 1991, 28 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399